A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems

Julien Forget, Frédéric Boniol, David Lesens, Claire Pagetti

A Real-Time Architecture Design Language for Multi-Rate Embedded Control Systems*

Julien Forget
ONERA, Toulouse, France
julien.forget@onera.fr

Frédéric Boniol
ONERA, Toulouse, France
frederic.boniol@onera.fr

David Lesens
EADS Astrium Space Transportation, Les Mureaux, France

Claire Pagetti
ONERA, Toulouse, France
claire.pagetti@onera.fr

ABSTRACT

This paper presents a language dedicated to the description of the software architecture of complex embedded control systems. The language relies on the synchronous approach but extends it to support efficiently systems with multiple real-time constraints, such as deadline constraints or periodicity constraints. It provides a high-level of abstraction and benefits from the formal properties of synchronous languages. The language defines a small set of rate transition operators, which enable the description of user-defined deterministic multi-rate communication patterns between components of different rates. The compiler of the language automatically translates a program into a set of communicating real-time tasks implemented as concurrent C threads that can be executed on a standard real-time operating system.

Categories and Subject Descriptors

C.3 [**Special purpose and application-based systems**]: Real-time and embedded systems; D.2.11 [**Software Architectures**]: Languages; D.3.2 [**Language classifications**]: Data-flow languages

General Terms

Languages; Design

Keywords

Real-time systems; Synchronous; Scheduling; Compilation;

1. INTRODUCTION

Our work is specifically targeted for *embedded control systems*. An embedded control system consists of a control loop including sensors, control algorithms and actuators that regulate the state of the system in its environment. Spacecraft and aircraft flight control systems are good examples of control systems. Their objective is to control the position, speed and attitude of the vehicle thanks to physical devices, such as control surfaces for an aircraft or thrusters for a spacecraft.

These systems are highly critical as a malfunction can lead to dramatic consequences. Thus, their implementation must be deterministic, not only functionally (producing *the right outputs* for given inputs) but also temporally (producing outputs *at the right time*). They are also often multi-periodic, as different devices of the system have different physical characteristics and must therefore be controlled at different rates. Finally, the system must respect deadline constraints corresponding to maximum end-to-end latencies between an observation (inputs) and the corresponding reaction (outputs).

1.1 Case study: The Flight Application Software

To motivate our work, we consider the programming of an adapted version of the Flight Application Software (FAS) of the Automated Transfer Vehicle (ATV) designed by EADS Astrium Space Transportation for resupplying the International Space Station (ISS). The FAS handles all the software functionalities of the system as long as no fault is detected. Its architecture is described informally in Fig. 1. Each operation is represented by a box and arrows between boxes represent data-dependencies. Arrows without sources represent system inputs and arrows without destination represent system outputs. Data-dependencies define a partial order between the different operations of the system, as the consumer of a data-flow must execute after the producer of the data-flow.

The FAS first acquires data: the orientation and speed from gyroscopic sensors (`GyroAcq`), the position from the GPS (`GPS Acq`) and from the star tracker (`Str Acq`) and telecommands from the ground station (`TM/TC`). The *Guidance Navigation and Control* function (divided into an upstream part, `GNC_US`, and a downstream part, `GNC_DS`) then computes the commands to apply while the *Failure Detection Isolation and Recovery* function (`FDIR`) verifies the state of the FAS and looks for possible failures. Finally, commands are computed and sent to the control devices: thruster orders for the *Propulsion Drive Electronics* (`PDE`), power distribution orders for the *Power System* (`PWS`), solar panel positioning orders for the *Solar Generation System*

*This work was funded by EADS Astrium Space Transportation

SAC'10 March 22-26, 2010, Sierre, Switzerland.

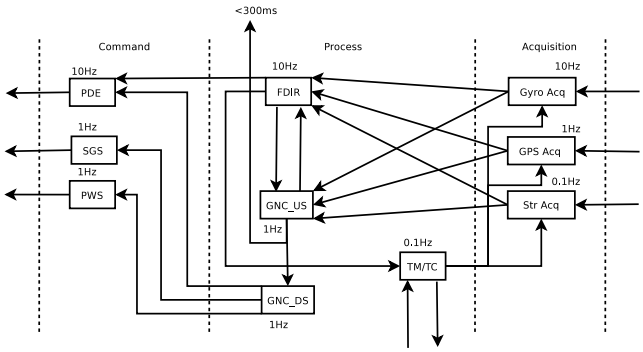

Figure 1: The Flight Application Software

(`SGS`) and telemetry towards the ground station (`TM/TC`), etc. Each operation has its own rate, ranging from 0.1Hz to 10Hz. An intermediate deadline constraint is imposed on data produced by the GNC Upstream (300ms while the period is 1s). Operations of different rates can communicate, which is an essential feature of such systems and has very important impacts on the complexity of the design and implementation process.

1.2 Contribution

Control systems like the FAS are very complex, their development involves several teams, which separately define the different functions of the system. The functions are then assembled by the integrator, who specifies the real-time properties of the functions and the communication schemes between the functions.

We propose a Real-Time Architecture Design Language designed to support this integration process. The language transposes the synchronous semantics [3] to the architecture design level. Defining the language as an ADL provides a high-level of abstraction, while relying on a synchronous semantics benefits from well-established formal properties. A program of this language consists of a set of *imported nodes*, implemented outside the program using other existing languages (C or Lustre for instance), and data-flows between the imported nodes. The language allows to specify multiple deadline and periodicity constraints on flows and nodes. It also defines a small set of rate transition operators that allows the programmer to precisely define the communication patterns between nodes of different rates. A program is automatically translated into a set of concurrent tasks implemented in C, which can be executed by standard real-time operating systems.

1.3 Paper Outline

This paper provides a complete presentation of the language and focuses on the illustration of its expressiveness through a series of examples. The formal semantics of the language is not detailed here and the compilation is only briefly described, due to lack of space. We first compare the language with existing work in Sect. 2. Then, we define the formal foundations of the language in Sect. 3. The main features of the language are detailed in Sect. 4. The expressiveness of the language is illustrated in Sect. 5. Finally, Sect. 6 gives an overview of the compilation process.

2. RELATED WORK

Probably the most widely used technique for programming real-time systems is to directly program the system with traditional imperative languages. These languages were not originally designed for real-time systems and the real-time aspects are mostly supported by Application Programming Interfaces (API) for real-time, which are very close to the underlying Real-Time Operating System (RTOS). For instance, the POSIX extensions for real-time [23] are a popular real-time API for C or Ada. Most real-time APIs share the same general principles: each operation of the system is first programmed as a separate function and functions are then grouped into threads, scheduled concurrently by the OS. As threads are related by data-dependencies, when dealing with critical systems the programmer must control the order in which data is produced and consumed (otherwise the behaviour of the system might not be predictable). This requires to implement synchronization mechanisms between threads, which can be a tedious and error-prone process.

This gradually leads programmers to consider languages with a higher-level of abstraction, which rely on an automated code generation process that translates the input program into lower level code. This greatly reduces the duration of the development process and produces more reliable software, as the correctness of the lower-level program is ensured by the translation process. The synchronous approach [3] proposes such a high level of abstraction, based on mathematical foundations, which make it possible to handle the compilation and the verification of a program in a formal way. This approach was originally mostly targeted for mono-periodic systems, ie systems where inputs and outputs are all acquired or produced at the same rate, but was later extended with real-time primitives, which enables the support of multi-periodic systems (different rates for different inputs/outputs). [6] supports the definition of deadline constraints in Esterel, but generates fully sequential code, which leads to a poor processor utilization factor in the case of multi-periodic systems. [12] defines primitives to specify multiple deadline and periodicity constraints on a Lustre program, but the compilation process is very specific to the target architecture (TTA). [1] extends this work and proposes a more generic compilation process, which generates a set of real-time tasks executable on a standard multi-task RTOS. Task communications are handled using the protocol proposed by [25], which ensures the conservation of the synchronous semantics of communicating tasks thanks to data-buffering mechanisms based on task priorities. Our language has a similar approach but focuses on the description of communication patterns between operations of different rates.

Architecture Description Languages (ADL) provide an additional layer of abstraction compared to synchronous languages, by focusing on the modelling of the interactions between high-level components of a system (typically, by assembling threads instead of functions). Threads specified in an ADL specification are considered as "black-boxes", implemented outside the ADL specification. AADL [14], CLARA [13], or GIOTTO [18] are specifically dedicated to the modelling of embedded real-time systems and describe a system as a set of communicating high-level real-time components. As far as we know, automated code generation is not supported by CLARA yet. Concerning AADL, the programmer only has a limited choice for the description of the communication schemes between components and cannot define its own schemes. This choice is even more limited in GIOTTO, where tasks can only consume data produced

before their release dates (instead of before the actual beginning of their execution), effectively leading to high end-to-end latencies. Finally, the MARTE profile for UML includes a powerful Clock Constraint Specification Language (CCSL) [2] for specifying the timing properties of a system. CCSL is far more expressive than our language, however it is targeted for system specification and system verification, not for automated code generation. Due to its expressivness, efficient code generation would most likely require to only consider a subset of the clock constraints provided by the language.

A possible approach to our problem would be to consider multi-periodic systems as a special case of Globally Asynchronous Locally Synchronous systems to reuse existing work [5, 19, 7, 8, 10]. However, multi-periodic systems are in essence completely synchronous, treating them as such allows to more clearly identify the real-time properties of a system (during the specification), provides better analysis possibilities (proving the correct synchronization of the operations, performing schedulability analysis) and allows the production of more optimized code (specific communication protocols).

Our language can be compared with other languages as follows:

- **ADLs**: The level of abstraction of the language is the same as that of existing ADLs but the language relies on a different semantics (the synchronous approach) and allows the programmer to define its own communication patterns between tasks. Furthermore, the complete compilation scheme is defined formally;
- **Synchronous languages**: Though the language has a synchronous semantics and shares similarities with Lustre [16], it addresses a different phase of the development (the integration) and enables efficient translation into a set of real-time tasks;
- **Imperative languages**: Though a program is in the end translated into (correct-by-construction) imperative code, the program is written in a data-flow style, which is easier to analyze and to verify.

3. SYNCHRONOUS REAL-TIME

This section presents the formal foundations of the language, which rely on the synchronous data-flow approach. The synchronous model is extended to support multi-rate systems more efficiently.

3.1 Synchronous Data-Flow

The synchronous approach simplifies the programming of real-time systems by abstracting from real-time and reasoning instead on a logical-time scale, defined as a succession of *instants*, where each instant corresponds to a reaction of the system. The synchronous hypothesis says that the programmer does not need to consider the dates at which computations take place during an instant, as long as the computations finish before the beginning of the next instant.

The synchronous approach has been implemented in several programming languages, our language is a synchronous data-flow language and is thus close to Lustre [16], Signal [4] or Lucid Synchrone [24]. Exactly as in Lustre, the variables and expressions of a program are *flows*, that is to say infinite sequences of values. The *clock* of a flow defines the instants during which the values of the flow must be computed. A program consists of a set of equations, structured into *nodes*. The equations of a node define its output flows from its input flows. A very simple program is given below:

```
imported node plus1(i:int) returns (o:int) wcet 5;
node N(i: int) returns (o: int)
let o=plus1(i); tel
```

The program first declares an imported node (`plus1`), implemented outside the program using other existing languages (C or Lustre for instance). The imported node has one integer input and one integer output. The duration of the node is also declared (this will be explained later). The program then defines a node `N`, with one input `i` and one output `o` and one equation. At each reaction of the program, the current value of the flow `o` is defined as the result of the application of `plus1` to the current value of the flow `i`. Imported node calls follow the usual data-flow semantics: an imported node cannot start its execution before all its inputs are available and produces all its outputs simultaneously at the end of its execution.

3.2 Relating Instants and Real-Time

In a classic synchronous data-flow program, the program describes the behaviour of the system for each of its basic iterations. Such an approach is not very well suited to the case of multi-periodic systems. Indeed, if we consider the system of Fig. 2, which consists of a fast operation `F` with a rate of 10Hz and a slow operation `S` with a rate of 2Hz, it is unclear what a basic iteration should consist of: one iteration of `F` and a varying part of `S` or one iteration of `S` and five iterations of `F` ?

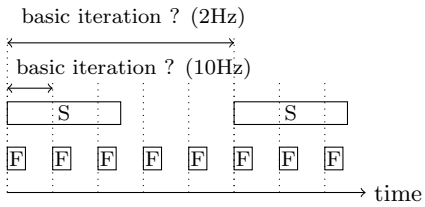

Figure 2: Base rate for a multi-periodic system

Therefore, we define a synchronous model that allows to relate instants to real-time. The main idea of this model is that a multi-periodic system can be considered as a set of locally mono-periodic synchronous processes assembled together to form a globally multi-periodic synchronous system. Locally, each process has its own logical time scale. When we assemble processes of different rates, we assemble processes of different logical time scales so we need a common reference to compare instants belonging to different logical time scales. This common reference is the real-time scale.

Our synchronous real-time model relies on the Tagged-Signal Model [20]. Given a set of *values* $\mathcal{V}$, a *flow* is a sequence of pairs $(v_i, t_i)_{i\in\mathbb{N}}$ where v_i is a value in $\mathcal{V}$ and t_i is a tag in $\mathbb{Q}$, such that for all i, $t_i < t_{i+1}$. The clock of a flow, is its projection on $\mathbb{Q}$. A tag represents an amount of time elapsed since the beginning of the execution of the program. Following the *relaxed synchronous hypothesis* of [12], at each activation a flow is required to be computed before its next activation. Thus, each flow has its own notion of instant and the duration of the instant t_i is $t_{i+1} - t_i$. Two flows are synchronous if the durations of their instants are the same. We can compare the durations of the instants of two

flows to determine if a flow is faster than the other and more importantly, we can determine *how much faster* this flow is.

3.3 Strictly Periodic Clocks

A clock is a sequence of tags, we define a particular class of clocks called *strictly periodic clocks* as follows:

DEFINITION 1. *(*Strictly periodic clock*). A clock* $h = (t_i)_{i\in\mathbb{N}}$, $t_i \in \mathbb{Q}$, *is* strictly periodic *if and only if:*

$$\exists n \in \mathbb{Q}^{+*},\ \forall i \in \mathbb{N},\ t_{i+1} - t_i = n$$

n is the period *of h, denoted $\pi(h)$ and t_0 is the* phase *of h, denoted $\varphi(h)$.*

A strictly periodic clock defines the real-time rate of a flow and is uniquely characterized by its phase and by its period:

DEFINITION 2. *The term $(n, p) \in \mathbb{Q}^{+*} \times \mathbb{Q}$ denotes the strictly periodic clock α such that: $\pi(\alpha) = n,\ \varphi(\alpha) = \pi(\alpha) * p$*

We then define clock transformations specific to strictly periodic clocks, which produce new strictly periodic clocks:

DEFINITION 3. *Let α be a strictly periodic clock, operations $/_., *_.$ and $\rightarrow_.$ are periodic clock transformations, that produce new strictly periodic clocks satisfying the following properties:*

$$\pi(\alpha/_. k) = k * \pi(\alpha),\ \varphi(\alpha/_. k) = \varphi(\alpha), k \in \mathbb{N}^*$$
$$\pi(\alpha *_. k) = \pi(\alpha)/k,\ \varphi(\alpha *_. k) = \varphi(\alpha), k \in \mathbb{N}^*$$
$$\pi(\alpha \rightarrow_. q) = \pi(\alpha),\ \varphi(\alpha \rightarrow_. q) = \varphi(\alpha) + q * \pi(\alpha), q \in \mathbb{Q}$$

Divisions and multiplications respectively decrease or increase the frequency of a clock while phase offsets shift the phase of a clock.

Strictly periodic clocks are actually a subset of the usual clocks of synchronous languages, which are defined using Boolean activation conditions. However, by restricting to this particular class of clocks, we are able to specify the real-time properties of a system more explicitly and to compile a program efficiently into a set of real-time tasks.

4. LANGUAGE DEFINITION

This section provides a detailed presentation of the language, focusing on its real-time primitives.

4.1 Real-time constraints

Real-time constraints are specified on node inputs and outputs, thus they represent constraints related to the environment of the node instead of constraints related to implementation concerns. First, the programmer can specify the period and the phase of an input/output as follows:

```
node periodic(i: int rate (10,1/2))
returns (o: int rate (10,0))
let ... tel
```

i has period 10 and phase $\frac{1}{2}$, so its clock sequence is (5, 15, 25, 35, ...) Time units correspond to whichever metric the underlying operating system uses for durations.

The programmer can also specify deadline constraints:

```
node deadline(i: int rate (10,0) before 2)
returns (o: int rate (10,0) due 8)
let ... tel
```

`x: due d` specifies that x has a deadline constraint of d. For each tag t_i in the clock of x, the computation of the value of x for this tag must complete before date $t_i + d$. `x: before d` specifies that x should be acquired by the program before the deadline d. In practice, this means that, before this deadline, the program must copy the value produced by the sensor to a local variable accessible to the other operations of the program.

Finally, the programmer must specify the duration of each imported node of the program. For instance, the following declaration specifies that the worst case execution time of `plus1` is 5 time units.

```
imported node plus1(i:int) returns (o:int) wcet 5;
```

4.2 Rate transition operators

We define a series of *rate transition operators*, based on periodic clock transformations. These operators enable the definition of user-specified communication patterns between nodes of different rates. First, the programmer can oversample or under-sample a flow periodically as follows:

```
node sampling(i: rate (10,0))
returns (o: rate (10,0))
  var vf, vs;
let (o, vf)=F(i,(0 fby vs)*^3); vs=S(vf/^3); tel
```

$e/\hat{}\ k$ only keeps the first value out of each k successive values of e. If e has clock α then $e/\hat{}\ k$ has clock $\alpha/_. k$. on the opposite, $e *\hat{} k$ duplicates each value of e, k times. If e has clock α, then $e *\hat{} k$ has clock $\alpha *_. k$. The `fby` operator is borrowed from LUCID SYNCHRONE. *cst* `fby` e first produces *cst* and then the values of e delayed by the period of e. The clock of *cst* `fby` e is the same as the clock of e. The behaviour of these operators is illustrated in Fig. 6

date	0	10	20	30	40	50	60	...
`vf`	vf_0	vf_1	vf_2	vf_3	vf_4	vf_5	vf_6	...
`vf/^3`	vf_0			vf_3			vf_6	...
`vs`	vs_0			vs_1			vs_2	...
`0 fby vs`	0			vs_0			vs_1	...
`(0 fby vs)*^3`	0	0	0	vs_0	vs_0	vs_0	vs_1	...

Figure 3: Periodic sampling operators

Three different operators based on clock phase offsets are available, as illustrated in the following program:

```
node offsets(i:rate(10,0))
returns (o:rate(10,1);p:rate(10,0);q:rate(10,1/2))
let
  o=tail(i); p=0::o; q=i^>1/2;
tel
```

`tail` (e) drops the first value of e. If e has clock α, then `tail` (e) has clock $\alpha \rightarrow_. 1$, so `tail` (e) becomes active one period later than e. *cst* `::` e produces *cst* one period earlier than the first value of e and then produces e. If e has clock α, then *cst* `::` e has clock $\alpha \rightarrow_. -1$. If e has clock α, $e \sim> q$ delays each value of e by $q * \pi(\alpha)$ (with $q \in \mathbb{Q}^+$). The clock of $e \sim> q$ is $\alpha \rightarrow_. q$. The behaviour of these operators is illustrated in Fig. 4.

date	0	5	10	15	20	25	30	...
`i`	i_0		i_1		i_2		i_3	...
`o=tail(i)`			i_1		i_2		i_3	...
`p=0::o`	0		i_1		i_2		i_3	...
`i~>2`		i_0		i_1		i_2		...

Figure 4: Phase offset operators

4.3 Polymorphic clocks

As we have seen in some of the previous examples, the types of the inputs and outputs of a node can be left unspecified. In this case, they will be inferred by the compiler. Similarly, clocks can be left unspecified, in which case they will be inferred by the compiler too. Furthermore, rate transition and phase offset operators can be applied to flows the rate of which is unspecified. For instance, the following node definition is perfectly valid:

```
node under_sample(i) returns (o)
let o=i/^2; tel
```

This program simply specifies that the rate of `o` is half that of `i` (ie twice the period of `o`), regardless of what the rate of `i` may be. The actual rate of the flows will only be computed when instantiating the node. The node can even be instantiated with different rates in the same program, for instance:

```
node poly(i: int rate (10, 0); j: int rate (5, 0))
returns (o, p: int)
let o=under_sample(i); p=under_sample(j); tel
```

The clock inferred for `o` is $(20, 0)$, while the clock inferred for `p` is $(10, 0)$. Such clock polymorphism increases the modularity with which systems can be programmed.

5. PROGRAMMING MULTI-RATE SYSTEMS

A key feature of the language is that it gives the programmer the freedom to choose the pattern used for the communications between operations of different rates, while most languages only allow a small set of predefined patterns. Rate transition operators apply very simple flow transformations but can be combined to produce various communication patterns. The formal definition of the language and the static analyses performed by the compiler ensure that only deterministic patterns are used.

We detail a series of communication patterns and show that common patterns can easily be defined as generic library operators that can be reused in different programs. We illustrate the different patterns on the simple example of Fig. 5, where a fast operation `F` (of period 10ms), exchanges data with a slow operation `S` (of period 30ms) and operation `F` exchanges data with the environment of the program.

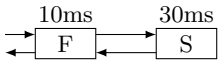

Figure 5: Simple multi-periodic communication loop

5.1 Sampling

The simplest communication pattern that can be implemented with our language is based on *data sampling*. For the example of Fig. 5, we obtain the node `sampling` presented previously in Sect. 4.2. Notice that we need a delay either from `S` to `F` or from `F` to `S`, otherwise there is a causality loop between `F` and `S`, the compiler cannot find an order for the execution of the program that respects all the data-dependencies. In the `sampling` version of the system, the delay avoids the reduction of the deadline of operation `S`. Let us now consider an alternative implementation:

```
node sampling2(i: rate (10, 0)) returns (o)
  var vf, vs;
let (o, vf)=F(i, vs*^3); vs=S((0 fby vf)/^3); tel
```

This version is also correct but an important difference with the previous version is that `S` must end before a relative deadline of 10ms, as it must end early enough for `F` to end before its own deadline (10ms). In the previous version, the delay avoids this restrictive deadline. The behaviours of these two examples are compared in Fig. 6.

date	0	10	20	30	40	50	60	...
vf	vf_0	vf_1	vf_2	vf_3	vf_4	vf_5	vf_6	...
vs	vs_0			vs_1			vs_2	...
vf/^3	vf_0			vf_3			vf_6	...
(0 fby vs)*^3	0	0	0	vs_0	vs_0	vs_0	vs_1	...
(0 fby vf)/^3	0			vf_2			vf_5	...
vs*^3	vs_0	vs_0	vs_0	vs_1	vs_1	vs_1	vs_2	...

Figure 6: Data sampling: the place of the delay

Using the operator `tail`, it is possible to choose which of the successive values of the fast operation should be consumed (ie not necessarily the first). For instance, we can consume the second value instead of the first:

```
node sampling_tail(i: rate (10, 0)) returns (o)
  var vf, vs;
let
  (o, vf)=F(i, (0::((0 fby vs)*^3)));
  vs=S((tail(vf))/^3);
tel
```

The behaviour of this program is illustrated in Fig. 7.

date	0	10	20	30	40	50	...
vf	vf_0	vf_1	vf_2	vf_3	vf_4	vf_5	...
tail(vf)/^3		vf_1			vf_4		...
vs		vs_0			vs_1		...
(0 fby vs)*^3		0	0	0	vs_0	vs_0	...
0::((0 fby vs)*^3)	0	0	0	0	vs_0	vs_0	...

Figure 7: Data sampling: starting the sample on the second value

5.2 Queuing

Combining rate transition operators and Lustre arrays, we can implement different communication patterns based on *data queuing*. First, we define a node that stores n successive values of a flow:

```
node store_n(const n:int;i,init) returns (A:int^n)
let
  A[n-1]=i;
  A[0..(n-2)]=(init^(n-1)) fby (A[1..(n-1)]));
tel
```

`int^n` denotes an integer array of size n. `A[n]` denotes the n^{th} value of array A. `init^(n-1)` denotes an array of size $n-1$ where each value of the array is set to *init*. `A[1..(n-1)]` denotes a slice of array A, from cell 1 to cell $n-1$. The behaviour of this node is the following. The n successive values of flow i are stored in the array A of size n. At each iteration, the node queues the current value of i at the end of A, moving the previous values of i by one cell towards the beginning of the array. At each instant, the array A contains the current value of i along with its $n-1$ previous values. The previous values are initialised to the value `init`. This is illustrated in Fig. 8.

i	i_0	i_1	i_2	i_3	...
A	$[0, 0, i_0]$	$[0, i_0, i_1]$	$[i_0, i_1, i_2]$	$[i_1, i_2, i_3]$	...

Figure 8: Storing successive values of a flow ($n = 3$, $init = 0$)

Then we define two more complex rate transition nodes:

```
node split(const n: int;i) returns(o)
var ifast;
let ifast=i*^n; o=ifast[count_n(n)]; tel

node join(const n: int; i,init) returns(o)
var ofast;
let ofast=store_n(n,i,init); o=ofast/^n; tel
```

The node `count_n` used in this example is a counter modulo n. The node `split` handles communications from slow to fast operations. It takes an array of size n as input and splits it into n successive values. The outputs are produced n times faster than the input. The node `join` handles communications from fast to slow operations. It joins n successive values of its input into an array. The output is n times slower than the input. This behaviour is illustrated below:

i	i_0	i_1	i_2	i_3	i_4	i_5	...
i'=join(3,i,0)	$[0,0,i_0]$			$[i_1,i_2,i_3]$			...
split(3,i')	0	0	i_0	i_1	i_2	i_3	...

We can then define a different version of the example of Fig. 5 that uses a queuing communication pattern:

```
node queuing(i: rate (10, 0)) returns (o)
  var vf, vs;
let
  (o, vf)=F(i, split(3, 0 fby vs));
  vs=S(join(3, vf, 0));
tel
```

This version requires flows `vf` and `vs` to be arrays of integers instead of simple integers and thus node `S` must return an array, not just an integer.

5.3 Mean value

We give a third example of communication pattern, which is an intermediate between sampling and queuing. Instead of queuing the successive values produced by the fast operations, we take their mean value:

```
node mean_n(const n: int;i, init) returns (m: int)
  var n_values, n_sum: int^n;
let
  n_values=store_n(n, i, init);
  n_sum[0]=n_values[0];
  n_sum[1..n-1]=n_values[1..n-1]+n_sum[0..n-2];
  m=(n_sum[n-1]/n)/^n;
tel
```

Notice that there is no causality loop in the third equation as each `n_sum[i]` depends on `n_sum[i-1]`, which are independent variables. The behaviour of this node is illustrated in Fig. 9.

i	1	2	3	4	5	...
n_values	[0, 0, 1]	[0, 1, 2]	[1, 2, 3]	[2, 3, 4]	[3, 4, 5]	...
n_sum	[0, 0, 1]	[0, 1, 3]	[1, 3, 6]	[2, 5, 9]	[3, 7, 12]	...
m	0			3		...

Figure 9: Computing the mean value of 3 successive "fast" values, (with $init = 0$)

We can then define a different version of the example of Fig. 5 that takes the mean value of 3 successive fast values (flow `vf`) and over-samples fast values (flow `vs`):

```
node sampling(i: rate (10, 0)) returns (o)
  var vf, vs;
let
  (o, vf)=F(i, (0 fby vs)*^3);
  vs=S(mean_n(3, vf, 0);
tel
```

5.4 A Complete Example

Fig. 10 gives a program for the FAS of Fig. 1 using communication patterns based on data sampling. Each operation of the system is first declared as an imported node. The program then specifies the wcet of each sensor and each actuator (for each input/output of the node `FAS`). The node `FAS` is the main node of the program. It first specifies the inputs (`gyro`, `gps`, `str`, `tc`) and outputs (`pde`, `sgs`, `gnc`, `pws`, `tm`) of the system with their rates (for instance `gyro` has period 100) and deadline constraints (for instance `gnc` has deadline 300). The body of the node then specifies the real-time architecture of the system. For instance, the first equation says that the node `Gyro_Acq` computes the variable `gyro_acq` from the input `gyro` and from the variable `tm`. As `tm` is produced with a period of 10s while `Gyro_Acq` has a period of 100ms, we over-sample `tm` by a factor 100. We use a delay before performing the over-sampling, to avoid reducing the deadline of node `TM_TC`. Some flows are over-sampled without using a delay, for instance when the outputs of the acquisition nodes are consumed by faster nodes (`gps_acq*^10` in the equation of the `FDIR` for instance). This implies that the acquisition nodes will have a deadline much shorter than their period, which means that they execute less frequently but they must respond fast. Communications from fast to slow nodes are handled using the under-sampling operator `/^`. For instance, the node `TM_TC` has a period of 10s and consumes the flow `fdir_tm`, which is produced with a period of 100ms, so we under-sample `fdir_tm` by a factor 100. Finally, in the equation of node `PWS` we apply a phase offset of one half to its input `gnc_pws`, which causes the node `PWS` to execute with a phase of half its period (it has period 1000 and phase 500).

6. COMPILATION

This section gives an overview of the compilation of a program.

6.1 Static Analyses

The compilation puts strong emphasis on the verification of the correctness of the program to compile. This consists of a series of static analyses, which are performed before code generation. The first analysis is the *type-checking*. The language is a strongly typed language, in the sense that the execution of a program cannot produce a run-time type error. The type-checking is fairly standard and follows [22].

The *causality check* verifies that the program does not contain cyclic definitions: a variable cannot instantaneously depend on itself (i.e. not without a `fby` in the dependencies). For instance, the equation `x=plus1(x);` is incorrect, it is similar to a deadlock since we need to evaluate `plus1(x)` to evaluate `x` and we need to evaluate `x` to evaluate `plus1(x)`. This analysis is similar to that of LUSTRE [17].

The *clock calculus* verifies that a program only combines flows that have the same clock. When two flows have the same clock, they are *synchronous* as they are always present at the same instants. Combining non-synchronous flows leads to non-deterministic programs as we access to undefined values. For instance we can only compute the sum of two synchronous flow, the sum of two non-synchronous flows is ill-defined (ambiguous). The clock calculus ensures that a synchronous program will never produce ill-defined values. The clock calculus of the language is more specific than other analyses. It extends the work of [11] and is defined

```
imported node Gyro_Acq(gyro, tc: int)
returns (o: int) wcet 3;
imported node GPS_Acq(gps, tc: int)
returns (o: int) wcet 3;
imported node Str_Acq(str, tc: int)
returns (o: int) wcet 3;
imported node FDIR(gyr, gps, str, gnc: int)
returns (to_pde, to_gnc, to_tm: int) wcet 15;
imported node GNC_US(fdir, gyr, gps, str: int)
returns (o: int) wcet 210;
imported node GNC_DS(us: int)
returns (pde, sgs, pws: int) wcet 300;
imported node TM_TC(from_gr, fdir: int)
returns (cmd: int) wcet 1000;
imported node PDE(fdir, gnc: int)
returns (pde_order: int) wcet 3;
imported node SGS(gnc: int)
returns (sgs_order: int) wcet 3;
imported node PWS(gnc: int)
returns (pws_order: int) wcet 3;
sensor gyro wcet 1; sensor gps wcet 1;
sensor str wcet 1; sensor tc wcet 1;
actuator pde wcet 1; actuator sgs wcet 1;
actuator gnc wcet 1; actuator pws wcet 1;
actuator tm wcet 1;

node FAS(gyro: rate (100,0); gps: rate (1000,0);
         str:  rate (10000,0); tc: rate (10000,0))
returns (pde, sgs;gnc: due 300; pws, tm)
var gyro_acq, gps_acq, str_acq, fdir_pde,
  fdir_gnc, fdir_tm, gnc_pde, gnc_sgs, gnc_pws;
let
  gyro_acq = Gyro_Acq(gyro, (0 fby tm)*^100);
  gps_acq = GPS_Acq(gps, (0 fby tm)*^10);
  str_acq = Str_Acq(str, 0 fby tm);
  (fdir_pde, fdir_gnc, fdir_tm) =FDIR(gyro_acq,
      gps_acq*^10, str_acq*^100, (0 fby gnc)*^10);
  gnc=GNC_US(fdir_gnc/^10, gyro_acq/^10, gps_acq,
      str_acq*^10);
  (gnc_pde, gnc_sgs, gnc_pws)=GNC_DS(gnc);
  pde = PDE(fdir_pde, (0 fby gnc_pde)*^10);
  sgs = SGS(gnc_sgs);
  pws=PWS(gnc_pws~>1/2);
  tm = TM_TC(tc, fdir_tm/^100);
tel
```

Figure 10: The FAS, with data-sampling

in details in [15]. If all the static analyses succeed, then the program has a well-defined, deterministic semantics and can be translated into lower-level code.

6.2 Muti-Task Code Generation

The compiler translates the program into C code. However, the translation differs from the classic "single-loop" sequential code generation, as we deal with multi-periodic systems. Let us consider the example below:

```
imported node A(i: int) returns (o: int) wcet 1;
imported node B(i: int) returns (o: int) wcet 5;
node multi(i: rate(3, 0)) returns (o)
var v;
let v=A(i); o=B(v/^3); tel
```

Out of three successive repetitions of `A`, only the first one must execute before a repetition of `B`. As shown in Fig. 11, a sequential execution of the program cannot respect all the deadline constraints of the program, the second repetition of operation `A` will miss its deadline (at date 6). We cannot find a sequence of execution for operations A and B that will respect the periodicity constraints of the two operations.

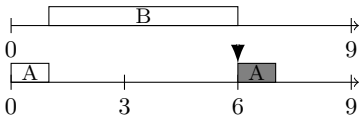

Figure 11: Sequential execution of node `multi`

However, if we implement operations A and B as two separate tasks and execute them concurrently with a preemptive RTOS, we can find a schedule for the execution of the tasks that respects the periodicity constraints of the two operations. Thanks to preemption, it is possible to start the execution of the second repetition of `A` earlier and thus to find a schedule that respects the deadline constraints of both tasks, as shown in Fig. 12. Marks on the time axis of B represent task preemptions and restorations: at date 3, B is suspended to execute the second repetition of A. After A completes, B is restored (at date 4) and resumes its execution where it stopped.

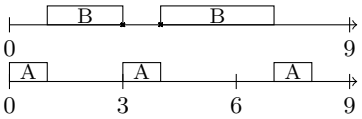

Figure 12: Preemptive scheduling of node `multi`

6.3 Overview of the Code Generation

The compiler translates the program into a set of communicating real-time tasks. Each imported node call of the program is translated into a task. The real-time attributes (T_i, C_i, r_i, d_i) of a task τ_i are obtained as follows: the period T_i is the period of the clock of the corresponding imported node, the initial release date r_i is the phase of the clock, the execution time C_i is the wcet specified for the imported node and the relative deadline d_i is either that specified for an input/output or by default equal to the period. For instance, the node `multi` described above is translated into two tasks: τ_A of attributes $(3, 1, 0, 3)$ and τ_B of attributes $(9, 5, 0, 9)$.

Tasks are related by data-dependencies, each task consumes data produced by other tasks (the inputs of the imported node) and produces data for other tasks (the outputs of the imported node). This implies precedence constraints between tasks: a task cannot start its execution before all the tasks that produce its inputs complete their execution and the task produces all its outputs simultaneously at its completion. A simple way to handle such inter-task communications is to allocate a buffer for each communication in a global memory, the producer of the data writes to this buffer when it executes and the consumer reads from this buffer when it executes. However, to preserve the semantics of the initial program, communications must respect two important properties. First, the producer must write before the consumer reads. Second, as we deal with periodic tasks, we must ensure that a new instance of the producer does not overwrite the value produced by its previous instance, if this previous value is still needed by some task.

To satisfy the first property, the compiler extends the technique proposed by [9] to the case of multi-periodic tasks. This consists in encoding task precedences by adjusting task real-time attributes. The deadline of each task is adjusted so as to be lower than that of its successors, and the release date of each task is adjusted so as to be higher that of its predecessors. The encoded task set can then simply be scheduled by the *Earliest-Deadline-First* policy [21]. As priorities are affected based on deadlines, each task will have a priority higher than that of its successors and lower than that of its predecessors, thus the first requirement is satisfied.

To satisfy the second property, the compiler implements a communication protocol based on data-buffering mechanisms, which ensure that the inputs of a task remain available until its deadline. The protocol is similar to that of [25] but is specifically targeted and optimized for multi-periodic systems. An important feature of this protocol is that it requires no specific synchronization primitives (semaphores).

7. CONCLUSION

We proposed a language for programming multi-rate embedded control systems. It transposes the synchronous semantics to an Architecture Design Language. It enables the description of a set of high-level communicating components with multiple deadline and periodicity constraints. The language focuses on the precise description of deterministic multi-rate communication patterns. The compiler translates a program into a set of communicating real-time tasks that respects the semantics of the original program. A prototype of the compiler has been implemented in OCaml (approximately 4000 lines of code). It generates a set of concurrent C threads and relies on the POSIX extensions for real-time to handle the real-time aspects.

8. REFERENCES

[1] M. Alras, P. Caspi, A. Girault, and P. Raymond. Model-based design of embedded control systems by means of a synchronous intermediate model. In *International Conference on Embedded Software and Systems (ICESS'09)*, Hangzhou, China, May 2009.

[2] C. André and F. Mallet. Combining CCSL and Esterel to specify and verify time requirements. In *ACM SIGPLAN/SIGBED 2009 Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'09)*, Dublin, Ireland, June 2009.

[3] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. In *Readings in hardware/software co-design*. Kluwer Academic Publishers, 2001.

[4] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the Signal language and its semantics. *Sci. of Compu. Prog.*, 16(2), 1991.

[5] G. Berry and E. Sentovich. Multiclock esterel. In *11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'01)*, Livingston, Scotland, Sept. 2001.

[6] V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. TAXYS=Esterel+Kronos. a tool for verifying real-time properties of embedded systems. In *40th IEEE Conference on Decision and Control*, volume 3, 2001.

[7] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli. Theory of Latency-Insensitive Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.*, 20(9), Sept. 2001.

[8] P. Caspi, C. Mazuet, and N. R. Paligot. About the design of distributed control systems: The quasi-synchronous approach. In *20th International Conference on Computer Safety, Reliability and Security (SAFECOMP'01)*, Budapest, Hungary, 2001.

[9] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2, 1990.

[10] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. *N*-Synchronous Kahn Networks: a relaxed model of synchrony for real-time systems. In *ACM International Conference on Principles of Programming Languages (POPL'06)*, Charleston, USA, Jan. 2006.

[11] J.-L. Colaço and M. Pouzet. Clocks as first class abstract types. In *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, USA, Oct. 2003.

[12] A. Curic. *Implementing Lustre Programs on Distributed Platforms with Real-Time Constraints*. PhD thesis, Université Joseph Fourier, Grenoble, 2005.

[13] S. Faucou, A.-M. Déplanche, and Y. Trinquet. An ADL centric approach for the formal design of real-time systems. In *Architecture Description Language Workshop at IFIP World Computer Congress (WADL'04)*, Aug. 2004.

[14] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical report, Carnegie Mellon University, 2006.

[15] J. Forget, F. Boniol, D. Lesens, and C. Pagetti. A multi-periodic synchronous data-flow language. In *11th IEEE High Assurance Systems Engineering Symposium (HASE'08)*, Nanjing, China, Dec. 2008.

[16] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proc. IEEE*, 79(9), 1991.

[17] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming (PLILP '91)*, Passau, Germany, 1991.

[18] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proc. IEEE*, 91(1), 2003.

[19] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for system design. Technical Report RR-4715, INRIA - Rennes, Feb. 2003.

[20] E. A. Lee and A. L. Sangiovanni-Vincentelli. Comparing models of computation. In *International Conference on Computer Aided Design (ICCAD'96)*, San Jose, USA, 1996.

[21] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.

[22] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, USA, 2002.

[23] POSIX.13. *IEEE Std. 1003.13-1998. POSIX Realtime Application Support (AEP)*. The Institute of Electrical and Electronics Engineers, 1998.

[24] M. Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, 2006.

[25] C. Sofronis, S. Tripakis, and P. Caspi. A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling. In *Sixth International Conference on Embedded Software (EMSOFT'06)*, Seoul, South Korea, 2006.